Performance Metrics for Object Oriented Front End Computers*

L.T. Hoff and J.F. Skelly, Brookhaven National Laboratory

Abstract

Front end computer software in the AGS and RHIC control systems makes extensive use of object oriented programming techniques. A conservative design approach to front end computer architecture has resulted in systems that are not performance limited; however, several interesting performance challenges have been identified. Performance metrics are presented for these situations. Programming techniques which foster high performance object oriented software are discussed.

1 WHY METRICS

The Object Oriented paradigm has been widely accepted for accelerator application software, but less so in front end computers. This situation may partially be due to skepticism about the ability of object oriented software to yield the performance necessary for the front end environment. This report presents some examples of the performance that can be accomplished with well designed object oriented software in a real time environment. These examples are followed by some tenets for good design of object oriented software for a performance-demanding environment.

2 RAPID DEVICE-OBJECT UPDATE AT A LINAC

The Linac injector at the AGS runs with a cycle rate between 5.0 and 7.5 Hz, providing proton beam to other users besides the AGS, and services multiple users within each AGS cycle; each pulse of the Linac is dedicated to one user, but the user may change from pulse to pulse, a feature called PPM (pulse-to-pulse modulation). Control of the Linac is implemented by 309 PPM devices, i.e. devices whose setpoint is user dependent; these devices must be sent a new user-dependent setpoint for each Linac cycle. In addition, another 109 non-PPM devices do not have user-dependent setpoints, but are read each cycle to update their measurements. Each of these 418 hardware devices is managed by a corresponding device object in the front end computer software.

These 418 devices are connected to the front end computer by 4 (Datacon) field buses of local design, similar to Mil-1553. The field bus implements a master-slave protocol; in each transaction, a command for the next Linac cycle is sent to a hardware device, and a rely is accepted which contains the measurement from the previous Linac cycle. The controller for these field buses is a VME card, which sends the commands and collects the replies using on-board memory, triggered by timing signals. The processor then reads the reply blocks from the memory on the field bus controller, and invokes each device object to parse the appropriate reply segment. This parsing operation permits each device object to update its measurement and status information, on a PPM basis, at the Linac cycle rate. Generation of reports for clients (i.e. application programs) of the front end computer is performed once per AGS cycle.

This activity was studied on a day when the AGS cycle was 6.2 seconds, and the Linac was executing 40 Linac cycles per AGS cycle. The device-object update rate was:

418 device-objects x 40 cycles / 6.2 sec =
2700 device-object-updates/sec

The vxWorks "spy" command was used to determine the processor loading. The task that performed the device-object updates consumed 20% to 25% of the processor cycles, and other tasks each consumed 0% to 2%. This front end computer utilizes a Motorola MVME162 processor with a 25 MHz 68040 cpu. Saturation of the cpu is projected to occur at about 10,000 device-object updates per second with this processor.

3 FACTORY IDIOM FOR AN EVENT SYSTEM

An event system is a mechanism for triggering the execution of tasks in an FEC, based on the occurrence of real-world events such as hardware interrupts, elapsed clock time, software signals, or signals from the accelerator timing system. It is desirable that the event system support rapid prototyping and rapid reconfiguration. This requirement mandates that the programming interface that associates an event with a software task should be independent of the type of event. For example, since accelerator signals may not be available during software development, clock events may be substituted during commissioning of the software.

Moreover, extensibility is also desirable, to permit easy addition of new types of events. The goal of the event system is to permit executing tasks of arbitrary complexity, not just software that can run in an interrupt service routine (ISR). Since the software must run at task level, not in an ISR, time must be budgeted for a task context switch between the triggering event and the execution of the software task. For the MVME162 processor board, this constrains event frequency to less than 40 kHz. Individual events anticipated for use with the event system had a maximum rate of 720 Hz, but higher aggregate rates within a single FEC must be tolerated by the event system.

* Work supported by U.S. Department of Energy

The criteria requiring flexibility and extensibility suggest the use of the "factory" or "virtual constructor" idiom [1]. Performance constraints resulting from adoption of this technique were evaluated.

The factory idiom is a technique for providing the benefits of inheritance and polymorphism while hiding the details of the derived classes from the code, which invokes the constructors. A base class is defined which implements a programming interface; derived classes add specialization but do not expand this interface. The base class also provides a method for making new objects, which are actually instances of the derived classes; this method uses context to determine which specific derived class to instantiate.

The event system base class provides methods for associating a software task with an event object, and a method for triggering a software task when the event occurs. The base class method which makes new objects relies on an extensible lookup table to associate context with a derived class, and provides a mechanism for derived classes to extend this table. Each derived class is responsible only for detecting the occurrence of its particular event, and for invoking the base class method that triggers the software task.

Whereas the instantiation of event objects with the factory idiom incurs additional overhead, the time-critical operation, triggering the software task, incurs little or no overhead relative to a procedural solution. Consequently this object oriented solution contributes substantial organizational flexibility and simplicity without compromising the realizable event trigger rate.

4 OBJECT ORIENTED PERFORMANCE TECHNIQUES

Books and courses are available to instruct the student in the proper use of object oriented languages, and it is neither desirable nor possible to cover that material here. But a modest collection of techniques will serve well to avoid the pitfalls that can prevent object oriented software from achieving its full performance potential. Many of these in fact are elementary principles that prudent programmers should employ in any case.

4.1 Orthodox Canonical Class Form

The "orthodox canonical class form"[2], also known as "value semantics", is simply a collection of five features that any well-designed class should provide. Absent these features, a class easily can be misused, quite unintentionally, in ways that silently degrade performance. With these features specified, such misuse is flagged by the compiler. These features are:

- Default constructor
- Copy constructor
- Destructor
- Conversion constructors
- Assignment operator

4.2 Object Oriented Analysis and Design

It is entirely possible to write object oriented software without conducting an object oriented analysis and design (OOAD) of the software project. But even elementary study of the project requirements may yield insight into the requirements which a class design should satisfy, and which lead to fundamental specifications of class design, such as:

- Should methods be declared public, protected or private.
- Should methods be declared virtual.
- Should data members be declared private or protected.

A proper OOAD addresses these issues, and ensures proper use of heap space, constructers, and the orthodox canonical class form.

4.3 Constructor Use

Object oriented languages offer the programmer substantial power, since the compiler does so much work behind the scenes. Certain constructors, specifically those which can be invoked with a single parameter, are used by the compiler as if they were conversion operators. The compiler may silently use these constructors in assignments or in subroutine invocations. Such constructors should receive special attention to ensure that their meaning as conversion operators is appropriate.

Another opportunity for silent invocation of constructors occurs with automatic variables. Inattention to the scope rules of automatic variables can likewise degrade performance.

4.4 Prudent Dynamic Memory Usage

Many C++ texts emphasize the “new” operator. This operator has important characteristics not found in procedural languages. However, this emphasis can be misconstrued to mean that dynamic memory allocation should be more commonly used in C++ than in procedural languages. Using heap memory, rather than stack or data memory incurs a performance penalty, and more easily results in memory “leaks”. However, if C++ classes support value semantics, then instances can be created as static, or automatic variables where appropriate, avoiding these pitfalls.

5 REFERENCES

[1] “Advanced C++ Programming Styles and Idioms”, James O. Coplien, Addison-Wesley Publishing Co. pp. 140-159

[2] ibid pp. 38-45